Comunicaciones en Red

En este apartado veremos como establecer una comunicación en red mediante sockets (protocolos TCP y UDP), y como acceder a URLs. Además Java permite mecanismos de comunicación por red de más alto nivel como por ejemplo RMI que veremos en capítulos posteriores.

1. Acceso a URLs

Una URL (*Uniform Resource Locator*) es una cadena utilizada para localizar un recurso en Internet. Dentro de la URL podemos distinguir varias componentes:

```
protocolo://servidor[:puerto]/recurso
```

Por ejemplo, en el caso de la dirección http://www.ua.es/es/index.html lo que se hará será acceder al servidor www.ua.es mediante protocolo **HTTP** y solicitar el recurso **/es/index.html**. El puerto por defecto es el 80, pero si el servidor Web atendiese en un puerto distinto a este deberíamos especificarlo también en la URL.

En Java tenemos el objeto **URL** que se encargará de representar las URLs. Podemos construir un objeto **URL** a partir del nombre completo de la URL:

```
URL url = new URL("http://www.ua.es/es/index.html");
```

Dado que muchas veces se especifican links relativos, será de ayuda contar con un segundo constructor que nos permita crear URLs a partir de la dirección base donde nos encontremos y de la dirección relativa solicitada:

```
URL url = new URL(direccion_base, direccion_relativa);
```

Aquí la dirección relativa puede referirse a un recurso alojado en el servidor donde nos encontremos o bien a un destino dentro de la web donde estamos, referenciado mediante **#nombre_destino**.

Existen más constructores de esta clase, permitiéndonos por ejemplo construir una URL dando cada elemento (protocolo, servidor, puerto, recurso) por separado. Siempre que creemos una URL deberemos capturar la excepción **MalformedURLException** que se producirá en el caso de estar mal construida.

```
try {
    URL url = new URL("http://www.ua.es/es/index.html");
```

```
} catch(MalformedURLException e) {
    System.err.println("Error: URL mal construida");
}
```

La clase **URL** proporciona métodos para obtener información sobre la URL que representa.

Para leer desde la dirección URL representada por el objeto deberemos obtener un flujo de entrada proveniente de ella. Para obtener este flujo utilizaremos el método **openStream()** del objeto **URL**.

```
InputStream in = url.openStream();
```

Una vez obtenido este flujo de entrada podremos leer de él o bien transformarlo a otro tipo de flujo como por ejemplo a un flujo de caracteres o de procesamiento.

Con esto será suficiente para leer desde una URL, pero también podremos establecer una conexión con la URL. Para ello deberemos utilizar el método **openConnection()** que nos devolverá un objeto del tipo **URLConnection**. Estableciendo una conexión podremos leer o escribir en la URL. Para ello deberemos:

1. Establecer la conexión
   URLConnection conn = url.**openConnection**();
2. Si vamos a escribir en ella, establecer la capacidad de salida con:
   conn.**setDoOutput**(true);
3. Si vamos a leer, obtener el flujo de entrada con:
   InputStream in = conn.**getInputStream**();
4. Si vamos a escribir, obtener el flujo de salida con:
   OutputStream out = conn.**getOutputStream**();
5. Leer o escribir en los flujos de entrada y de salida obtenidos como vimos en capítulos anteriores.

2. Comunicación por SOCKETS

El mecanismo de acceso a URLs proporcionado por Java nos permite establecer conexiones en red de alto nivel, sin embargo es posible que necesitemos establecer una conexión a bajo nivel mediante protocolos TCP o UDP. Los *sockets* nos permitirán realizar este tipo de conexiones en red de bajo nivel, estableciéndose un canal de comunicación entre un par de *sockets*, cada uno de los cuales puede residir en una máquina distinta.

Los *sockets* son los extremos de un canal de flujo de datos a través de la red. Estos *sockets* vendrán representados en Java por el objeto **Socket**, que nos permitirá conectar con una máquina remota y abrir canales de E/S para comunicarnos con ella. Además tenemos un tipo especial de *socket* en el servidor, en la clase **ServerSocket**, que se quedará a la escucha en un determinado puerto de la máquina servidora y cuando le llegue una petición de conexión de un cliente creará un objeto **Socket** para atenderlo.

Para establecer una conexión con el servidor, desde el cliente deberemos crear un objeto **Socket** proporcionando el nombre de la máquina (host o dirección IP) y el puerto al que nos vamos a conectar:

```
Socket cliente = new Socket("127.0.0.1", 6006);
```

En este ejemplo nos conectaremos a nuestra propia máquina local mediante su dirección de *loopback*, al puerto 6006. Si en nuestra máquina tenemos un servidor atendiendo en dicho puerto se establecerá una comunicación con él.

Para obtener los canales de E/S asociados a dicho *socket* podremos utilizar los métodos **getInputStream()** y **getOutputStream()**, y convertir posteriormente estos flujos de bytes a otros tipos para mayor comodidad en la lectura y escritura.

```
InputStream in = cliente.getInputStream();
OutputStream out = cliente.getOutputStream();
```

Una vez terminemos de usar el *socket*, aparte de cerrar los flujos de entrada y salida que hayamos utilizado deberemos cerrar la conexión del *socket*con:

```
cliente.close();
```

3. Creación de un servidor

En el servidor deberemos crear un objeto **ServerSocket** enlazado a un determinado puerto que se quede a la espera de peticiones de conexión por parte de los clientes. Deberemos proporcionar el constructor de dicha clase el puerto en el que escuchará el *socket* servidor.

```
ServerSocket server = new ServerSocket(6006);
```

Para hacer que permanezca a la espera de una petición de conexión utilizaremos en método *accept()*. De esta manera el programa servidor quedará bloqueado en este método a la espera de la llegada de una

petición de conexión. Cuando llegue esta petición se devolverá un objeto **Socket** que se utilizará para la comunicación con el cliente.

```
Socket sock = server.accept();
```

Este *socket* que devuelve deberá ser utilizado al igual que los *sockets* utilizados en el cliente.

4. Servidor para múltiples clientes

Si queremos hacer que un servidor pueda atender a varios clientes simultáneamente, tras aceptar una conexión deberá quedarse nuevamente a la espera de nuevas peticiones, al mismo tiempo que está atendiendo la conexión que ha aceptado previamente. Para poder realizar estas dos tareas simultáneamente deberemos utilizar hilos. El bucle típico de espera de conexiones en el servidor será como el siguiente:

```
Thread t;
Socket sock;
while(true) {
    sock = server.accept();
    t = new HiloServidor(sock);
    t.start();
}
```

La clase **HiloServidor** será una clase definida por el usuario que herede de la clase **Thread** y que en el método **run()** contendrá el código para atender a cada cliente. Al proporcionar el **Socket** asociado a cada cliente en el constructor, cada hilo se comunicará con el *socket* que se especificó en el momento de su construcción, permitiendo así que se ejecuten un número indefinido de hilos simultáneamente cada uno de ellos conectado a un cliente distinto.

5. Comunicación por datagramas y por circuitos virtuales

La forma de comunicación por *sockets* que hemos visto es la denominada comunicación por circuitos virtuales, que sigue el protocolo **TCP**. En este tipo de comunicación se crea un canal permanente para la circulación de los datos, de forma que se garantiza de los datos enviados no se perderán y llegarán en el mismo orden en que se enviaron. Sin embargo este mecanismo de comunicación tiene el inconveniente de consumir más recursos al tener que crear un canal permanente.

Otro mecanismo de comunicación que consume menos recursos es la comunicación por datagramas, que sigue el protocolo **UDP**. En este

caso cada paquete (datagrama) enviado sigue su propio camino por la red, por lo que puede ser que lleguen a su destino desordenados, pero no mantienen un canal abierto permanentemente consumiendo recursos. Tampoco se garantiza que el datagrama llegue a su destino.

En caso de un tráfico intenso de datos será conveniente la comunicación por circuitos virtuales, mientras que cuando el tráfico sea escaso, y la ordenación o la pérdida de paquetes no sean críticas, convendrá utilizar datagramas.

En el caso de los datagramas tendremos un único tipo de *socket*: **DatagramSocket**. Podremos proporcionar en el constructor el puerto en el que debe permanecer escuchando:

```
DatagramSocket sock = new DatagramSocket(6006);
```

Si utilizamos el constructor vacío el sistema será el encargado de asignar el puerto en el que escuchará el *socket*.

```
DatagramSocket sock = new DatagramSocket();
```

En este caso vemos que no existen cliente y servidor claramente definidos, pero al menos uno de los dos *sockets* que vayamos a comunicar deberá especificar el puerto en el que atiende para que el otro sea capaz de saber a qué puerto debe conectarse. El *socket* del que especificamos el puerto podrá ser considerado servidor, mientras que el *socket* que envía información por primera vez conociendo el puerto que hemos especificado será el cliente.

Vemos en este caso que en la creación del *socket* no se ha especificado ninguna dirección a la que conectarse, ya que el socket no establece ninguna conexión. La dirección de destino deberá ser especificada en cada datagrama (paquete). Estos datagramas están definidos en la clase**DatagramPacket**. Estos paquetes se construirán a partir de un buffer (array de bytes) que se utilizará para almacenar los datos recibidos o los datos a enviar:

```
DatagramPacket paquete_recibido =
new DatagramPacket(buffer, buffer.length);
```

Cuando sea un paquete para ser enviado, además del buffer deberemos especificar la dirección y puerto de destino:

```
DatagramPacket paquete_enviar =
    new DatagramPacket(buffer, buffer.length, direccion,
puerto);
```

La dirección deberá ser especificada como un objeto **InetAddress**.

Para enviar un datagrama utilizaremos en método *send()* del *socket*:

```
sock.send(paquete_enviar);
```

Cuando queramos permanecer a la espera de recibir un paquete utilizaremos el método *receive()*:

```
sock.receive(paquete_recibido);
```

Una vez recibido un paquete podremos obtener el puerto desde el que nos lo envió el remitente mediante el método *getPort()*, de esta forma sabremos a que dirección debemos contestar.

```
int puerto = paquete_recibido.getPort();
```